

Finding Submasses in Weighted Strings with Fast Fourier Transform

Nikhil Bansal^a, Mark Cieliebak^b, Zsuzsanna Lipták^{c,*}

^a*IBM Research, T.J. Watson Research Center*

^b*Institute of Theoretical Computer Science, ETH Zurich, and Center for Web Research, Department of Computer Science, University of Chile*

^c*AG Genome Informatics, Technical Faculty, University of Bielefeld*

Abstract

We study the Submass Finding Problem: Given a string s over a weighted alphabet, i.e., an alphabet Σ with a weight function $\mu : \Sigma \rightarrow \mathbb{N}$, we refer to a mass $M \in \mathbb{N}$ as a *submass* of s if s has a substring whose weights sum up to M . Now, for a set of input masses $\{M_1, \dots, M_k\}$, we want to find those M_i which are submasses of s , and return one or all occurrences of substrings with mass M_i . We present efficient algorithms for both the decision and the search problem. Furthermore, our approach allows us to compute efficiently the number of different submasses of s .

The main idea of our algorithms is to define appropriate polynomials such that we can determine the solution for the Submass Finding Problem from the coefficients of the product of these polynomials. We obtain very efficient running times by using Fast Fourier Transform to compute this product. Most of our algorithms run in time $\mathcal{O}(\mu_s \log \mu_s)$, where μ_s is the total mass of string s . Employing methods for compressing sparse polynomials, this runtime can be viewed as $\mathcal{O}(\sigma(s) \log^2 \sigma(s))$, where $\sigma(s)$ denotes the number of different submasses of s . This allows us to obtain algorithms that have running time as a function of $\sigma(s)$. Our algorithms give the best known running times, if $\sigma(s)$ is small. However, in general $\sigma(s)$ could be as large as $O(n^2)$, in which case our algorithms are not necessarily optimal.

Key words: string algorithms, weighted strings, protein identification, Fast Fourier Transform

* Corresponding author. Universität Bielefeld, AG Genominformatik, Technische Fakultät. Postfach 10 01 31, D-33592 Bielefeld, Germany.

Email addresses: nikhil@us.ibm.com (Nikhil Bansal), cieliebak@inf.ethz.ch (Mark Cieliebak), zsuzsa@cebitec.uni-bielefeld.de (Zsuzsanna Lipták).

1 Introduction

Over the past few years, interest in the area of weighted strings has received increasing attention. A *weighted string* is defined over an alphabet $\Sigma = \{a_1, \dots, a_{|\Sigma|}\}$ with a weight function $\mu : \Sigma \rightarrow \mathbb{N}$, which assigns a specific weight (or mass) to each character of the alphabet. The weight of a string s is just the sum of the weights of all characters in s .

Several applications from bioinformatics can be formalized as problems on strings over a weighted alphabet; most notably, mass spectrometry experiments, which constitute an experimentally very efficient method of protein identification and de-novo peptide sequencing. Mass spectrometry is also increasingly being used for DNA molecules. For our purposes, proteins are strings over the 20-letter amino acid alphabet, and DNA molecules are strings over the alphabet of the four bases. The molecular masses of the amino acids and the DNA bases are known up to high precision. In order to enforce that the masses be positive integers, we assume that non-integer masses have been scaled.

One of the main applications of protein mass spectrometry is database lookup. Here, a protein is broken up into substrings, the molecular masses of the substrings are determined, and the list of masses is compared to a protein database. The latter step gives rise to the mass finding problems that we study in this paper.

Definitions and Problem Statements

We first fix some notation for weighted strings. Let Σ be a finite alphabet with a mass function $\mu : \Sigma \rightarrow \mathbb{N}$, where we denote by \mathbb{N} the set of positive integers excluding 0. We refer to such an alphabet as a *weighted alphabet*.¹ We denote by $\mu_{\max} = \max \mu(\Sigma)$, the largest mass of a single character. For a string $s = s_1 \dots s_n$ over Σ , define $\mu(s) := \sum_{i=1}^n \mu(s_i)$. We denote the length n of s by $|s|$. We call $M > 0$ a *submass* of s if there exists a substring t of s with mass M , or, equivalently, if there is a pair of indices (i, j) such that $\mu(s_i \dots s_j) = M$. We call such a pair (i, j) a *witness* of M in s , and we denote the number of witnesses of M in s by $\kappa(M) = \kappa(M, s)$. Note that $\kappa(M) \leq n$. Finally, we denote by $\sigma(s)$ the number of different submasses of string s . Note that for any string s , the number of different submasses $\sigma(s) \geq n$ and $\sigma(s) \leq n(n+1)/2$ and $\sigma(s) \leq \mu_{\max} n$.

We want to solve the following problems:

¹ Note that we use the expressions “weight” and “mass” synonymously, hence “weighted alphabet” but “mass function.”

Submass Query Problem Fix a string s over Σ . Let $|s| = n$.

INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.

OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ such that $i \in I$ if and only if M_i is a submass of s .

Submass Witness Problem Fix a string s over Σ . Let $|s| = n$.

INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.

OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ such that $i \in I$ if and only if M_i is a submass of s , and a set $\{(b_i, e_i) : i \in I, (b_i, e_i) \text{ is a witness of } M_i \text{ in } s\}$.

Submass All Witnesses Problem Fix a string s over Σ . Let $|s| = n$.

INPUT: k masses $M_1, \dots, M_k \in \mathbb{N}$.

OUTPUT: A subset $I \subseteq \{1, \dots, k\}$ such that $i \in I$ if and only if M_i is a submass of s , and for each $i \in I$, the set of all witnesses $W_i := \{(b, e) : (b, e) \text{ is witness of } M_i \text{ in } s\}$.

Simple Solutions

The three problems above can be solved by one of several simple algorithms that we now describe. The first algorithm, which we refer to as `LINSEARCH`, moves two pointers along the string, one pointing to the potential beginning and the other to the potential end of a substring with mass M . The right pointer is moved if the mass of the current substring is smaller than M , the left pointer, if the current mass is larger than M . The algorithm solves each problem in $\mathcal{O}(kn)$ time and uses $\mathcal{O}(1)$ space in addition to the storage space required for the input string and the output.

Another simple algorithm, `BINSEARCH`, computes all submasses of s in a preprocessing step and stores them in a sorted array, which can then be queried in time $\mathcal{O}(k \log n)$ for k input masses for the `SUBMASS QUERY PROBLEM` and the `SUBMASS WITNESS PROBLEM`. The storage space required is proportional to $\sigma(s)$, the number of different submasses of string s , and is thus $\mathcal{O}(n^2)$, while the preprocessing time is $\Theta(n^2 \log \sigma(n))$. For the `SUBMASS ALL WITNESSES PROBLEM`, we need to store in addition all witnesses, requiring space $\Theta(n^2)$; in this case, the query time becomes $\mathcal{O}(k \log n + K)$, where $K = \sum_{i=1}^k \kappa(M_i)$ is the number of witnesses for the query masses. Note that any algorithm solving the `SUBMASS ALL WITNESSES PROBLEM` will have runtime $\Omega(K)$.

Alternatively, we can use a Boolean array of size μ_s for storing all submasses of s , thus allowing constant time access for queries. Then the query running time becomes $\Theta(k)$ and the storage space $\Theta(\mu_s)$. For the `SUBMASS QUERY PROBLEM` and the `SUBMASS WITNESS PROBLEM`, this yields time $\Theta(k)$ and space $\Theta(\mu_s)$, and for the `SUBMASS ALL WITNESSES PROBLEM`, $\Theta(\max(k, K))$ time and $\Theta(\max(\mu_s, n^2))$ space.

In general, these results are of the following type: Either they have a high preprocessing time and a low query time, or they have a low preprocessing time, then they have a high query time for each query. Our goal is find algorithms that have both a low preprocessing time and a low query time. By low processing time, we mean a preprocessing that beats the running time of the previous algorithms in the case when $\sigma(s)$ is substantially smaller than n^2 . Later, we will give evidence for why this is essentially the best we could hope for.

Results

In this paper, we present a novel approach to the problems above which often outperforms the naïve algorithms. The main idea is similar to using generating functions for counting objects, which have been applied, for instance, in attacking the Coin Change Problem [1]. Instead of infinite polynomials though, we use finite ones as follows. We define appropriate polynomials such that we can determine the solution for the three problems above from the coefficients of the product of these polynomials. We will obtain very efficient running times by using Fast Fourier Transform to compute this product. More precisely, ALGORITHM 1 solves the SUBMASS QUERY PROBLEM with preprocessing time $\mathcal{O}(\mu_s \log \mu_s)$, query time $\mathcal{O}(k \log n)$, and storage space $\Theta(\sigma(s))$. For the SUBMASS WITNESS PROBLEM, we present a Las Vegas algorithm, ALGORITHM 2, with preprocessing time $\mathcal{O}(\mu_s \log^3 \mu_s)$, expected query time $\mathcal{O}(k \log n)$, and storage space $\Theta(\sigma(s))$. Finally, we present ALGORITHM 3, a deterministic algorithm for the SUBMASS ALL WITNESSES PROBLEM with total running time $\mathcal{O}((Kn\mu_s \log \mu_s)^{\frac{1}{2}})$, where K is the output size, i.e., the total number of witnesses.

Our results above are stated in terms of μ_s , the total mass of the string s . However, we can use the sparse polynomial multiplication technique of Cole and Hariharan [2] in a straightforward way to give Las Vegas variants of our algorithms, where each term μ_s in the expected running time can be replaced by $\sigma(s) \text{polylog}(\sigma(s))$. We give a brief summary of the result: Given two vectors v_1 and v_2 of length n , comprising only of non-negative entries. Let w denote their product, and $\|w\|$ the number of non-zero entries in w . In [2] it is shown how to obtain the non-zero entries of w in time $O(\|w\| \log^2 n)$, using a Las Vegas randomized algorithm whose failure probability is inverse polynomial in n . This algorithm uses the idea of hashing the original entries in the range $\alpha\|w\|$ for a suitable constant α , so as not to have too many collisions, and then performing $O(\log n)$ independent runs of the algorithms. The proper value of $\|w\|$ is determined by the standard guessing and doubling trick. Thus, throughout this paper we present our runtimes as a function of μ_s with the understanding that μ_s is identical to $\sigma(s)$ up to polylogarithmic factors.

To compare our results with previous results, note that if μ_{\max} is a constant

(and hence $\sigma(s)$ is $O(n)$), then the running times for our algorithms are significantly better than the previously known naive algorithms. Infact, most results are better even if μ_{\max} is allowed to be a function of the string length n . For example, for the SUBMASS QUERY PROBLEM our algorithm has preprocessing time of $O(\sigma(s)\text{polylog}(\sigma(s)))$, while the naive algorithms (which has similar query time as our algorithm) require $O(n^2)$ preprocessing time. Thus our algorithm performs better whenever $\mu_{\max} = o(n^{1/(\text{polylog}(n))})$. In situations where μ_{\max} one can view our results in a simpler way by substituting $\sigma(s) = O(n)$. However, we express our results more generally in terms of $\sigma(s)$.

Later we will show that our algorithmic technique can be trivially adapted to solve the following problem in $O(\sigma(s)\log(\sigma(s)))$ time.

Number of Submasses Problem Given string s of length n , how many different submasses does s have?

This is interesting because it is a major open whether this problem can be solved in $o(n^2)$ time in general when $\sigma(s) = \Theta(n^2)$ (see the discussion in Section 5). This suggests that $\sigma(s)$ is perhaps the right quantity to measure the running time of our algorithms. In particular, it seems very hard to obtain algorithms that have improved running times than the naive ones in the general case.

It should be pointed out that for real-life mass spectrometry applications with today's technology, our algorithms are unlikely to outperform the naïve algorithms, in particular BINSEARCH. Our experiments show that the number of submasses $\sigma(s)$ is quadratic in the string length for real protein strings. We tested approximately 5000 protein strings and domains from the protein data base SCOP [3,4], with lengths between 50 and 1500 amino acids, and precisions of 0.1 Da and 0.01 Da (thus, the scaling factors to obtain integers were 10 and 100, respectively). The total mass of the strings was, as can be expected, approximately $100 \cdot (1/\text{precision}) \cdot n$, where n is the length of the string; the average mass of one amino acid is around 100 Da. We found that $\sigma(s)$ was always above $\frac{1}{2} \frac{n(n+1)}{2}$, i.e., more than half the maximum possible number of witnesses. Obviously, in this case, the computational overhead of our algorithms yields running times that are much larger than that of the simple BINSEARCH.

For DNA strings with lengths of several hundreds of thousands of bases, on the other hand, this will no longer be the case, since then the number of submasses can be significantly smaller than n^2 . Although current mass spectrometry technology for DNA only allows measurements of up to a few thousand Da (this means that large submasses, corresponding to long substrings, cannot be measured at present), the technology is developing at a rapid pace, and we trust that the ideas and algorithms presented in this paper will be applicable

in the not-so-far future.

Related Work

Several simple algorithms for the SUBMASS QUERY PROBLEM were presented in [5], including LINSEARCH and BINSEARCH (which we adapted here straightforwardly to solve the other two problems as well). Furthermore, an algorithm was presented which solves the SUBMASS WITNESS PROBLEM for *one* query with $\mathcal{O}(n)$ storage space and query time $\mathcal{O}(\frac{n}{\log n})$, using $\mathcal{O}(n)$ time and space for preprocessing. This algorithm can, of course, be used for k queries, yielding an overall runtime of $\mathcal{O}(n + \frac{kn}{\log n})$. However, this is an asymptotic result only, since the constants in this running time are so large that for a 20-letter alphabet and realistic string sizes, the algorithm is not applicable. For binary alphabets, another algorithm was presented which solves the SUBMASS QUERY PROBLEM for one query with $\mathcal{O}(n)$ space and query time $\mathcal{O}(\log n)$ but does not produce witnesses.

Another algorithm for the SUBMASS ALL WITNESSES PROBLEM preprocesses the database by compressing witnesses using suffix trees [6]. This algorithm works only under the assumption that the queries are limited in range.

In the context of database lookup for proteins, the fragmentation of a protein is usually done using a site-specific cleavage enzyme, most commonly trypsin, which cuts after each amino acid arginine (one-letter-code R) and lysine (K), unless followed by a proline (P). In this case, only those submasses need to be considered that have witnesses where either the first and last characters or flanking substrings are known. From a theoretical point of view, this is an easy problem, since all such submasses along with their witnesses can be computed in a straightforward preprocessing step. This technique is implemented in software tools such as Sequest [7]. On the other hand, a random digestion model is increasingly used, where breaking points are not known in advance. This model has been studied recently in [6,8,5], and is appropriate for instance for collision induced dissociation (e.g. using argon or helium), or where several enzymes are applied at the same time.

The study of weighted strings and their submasses has further applications in those problems on strings over an un-weighted alphabet where the focus of interest are not substrings, but rather equivalence classes of substrings defined by multiplicities of characters. One examines objects of the form $(n_1, \dots, n_{|\Sigma|})$ which represent all strings $s_1 \dots s_n$ such that the cardinality of character a_i in each string is exactly n_i , for all $1 \leq i \leq |\Sigma|$. These objects have been referred to in recent publications variously as *compositions* [9], *compomers* [10,11], *Parikh-vectors* [12], *multiplicity vectors* [5], and *π -patterns* [13]. A similar approach has been referred to as *Parikh-fingerprints* [14,15]. Here, Boolean vectors are considered of the form $(b_1, \dots, b_{|\Sigma|})$, where $b_i = 1$ if and only if a_i occurs in

the string. Applications range from identifying gene clusters [15] to pattern recognition [14], alignment [9] or SNP discovery [11].

2 Searching for Submasses Using Polynomials

In this section, we introduce the main idea of our algorithms, the encoding of submasses via polynomials. We first prove some crucial properties, and then discuss algorithmic questions.

Let $s = s_1 \dots s_n$. In the rest of the paper, we denote by μ_s the total mass of the string s , and the empty string by ε . Define, for $0 \leq i \leq n$,

$$p_i := \sum_{j=1}^i \mu(s_j) = \mu(s_1 \dots s_i),$$

the i 'th prefix mass of s . In particular, $p_0 = \mu(\varepsilon) = 0$. We define two polynomials

$$P_s(x) := \sum_{i=1}^n x^{p_i} = x^{\mu(s_1)} + x^{\mu(s_1 s_2)} + \dots + x^{\mu_s}, \quad (1)$$

$$Q_s(x) := \sum_{i=0}^{n-1} x^{\mu_s - p_i} = x^{\mu_s} + x^{\mu_s - \mu(s_1)} + \dots + x^{\mu_s - \mu(s_1 \dots s_{n-1})} \quad (2)$$

Now consider the product of $P_s(x)$ and $Q_s(x)$,

$$C_s(x) := P_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} c_m x^m. \quad (3)$$

Since any submass of s with witness (i, j) can be written as a difference of two prefix masses, namely as $p_j - p_{i-1}$, we obtain the following

Lemma 2.1 *Let $P_s(x), Q_s(x)$ and $C_s(x)$ from Equations (1) through (3). Then for any $m \leq \mu_s$, $\kappa(m) = c_{m+\mu_s}$, i.e., the coefficient $c_{m+\mu_s}$ of $C_s(x)$ equals the number of witnesses of m in s .*

Proof: By definition, we have

$$\begin{aligned} C_s(x) &= P_s(x) \cdot Q_s(x) = \sum_{j=1}^n x^{p_j} \sum_{i=0}^{n-1} x^{\mu_s - p_i} = x^{\mu_s} \cdot \sum_{1 \leq i, j \leq n} x^{p_j - p_{i-1}} \\ &= x^{\mu_s} \cdot \sum_{1 \leq i \leq j \leq n} x^{\mu(s_i \dots s_j)} + x^{\mu_s} \cdot \sum_{1 \leq j < i \leq n} x^{-\mu(s_{j+1} \dots s_{i-1})}. \end{aligned}$$

Let $[x^i]A(x)$ denote the coefficient a_i of x^i of the polynomial $A(x) = \sum_j a_j x^j$. Then, for any $m \leq \mu_s$,

$$\begin{aligned} \kappa(m) &= |\{(i, j) : \mu(s_i \dots s_j) = m\}| = [x^m] \left(\frac{1}{x^{\mu_s}} C_s(x) \right) \\ &= [x^{m+\mu_s}] C_s(x) = c_{m+\mu_s}. \end{aligned}$$

□

For a proposition Π , we denote by $[\Pi]$ the Boolean function which equals 1 if Π is true, and 0 otherwise.² Lemma 2.1 immediately implies the following facts.

Corollary 2.2 *For $C_s(x)$ from (3), $\sum_{m=\mu_s+1}^{2\mu_s} [c_m \neq 0] = \sigma(s)$, the number of submasses of s . Furthermore, $\sum_{m=\mu_s+1}^{2\mu_s} c_m = \frac{n(n+1)}{2}$.*

Thus, polynomial C_s also allows us to compute the number of submasses of s .

Example 1 *Let $s = baac$, $\mu(a) = 2, \mu(b) = 3, \mu(c) = 5$. Then $P_s(x) = x^3 + x^5 + x^7 + x^{12}$, $Q_s(x) = x^{12} + x^9 + x^7 + x^5$, $C_s(x) = x^8 + 2x^{10} + 3x^{12} + 2x^{14} + x^{15} + x^{16} + 2x^{17} + 2x^{19} + x^{21} + x^{24}$. Dividing the terms $c_i x^i$ with $i > 12$ by $x^{12} = x^{\mu_s}$ yields $2x^2 + x^3 + x^4 + 2x^5 + 2x^7 + x^9 + x^{12}$. This yields the submasses 2, 3, 4, 5, 7, 9, 12 with two witnesses for 2, 5, and 7, and one witness for each of the other submasses.*

2.1 Algorithm and Analysis

We now present an algorithm to solve the SUBMASS QUERY PROBLEM and the NUMBER OF SUBMASSES PROBLEM. The algorithm primarily consists of computing polynomial $C_s(x)$.

ALGORITHM 1

- (1) Preprocessing step:
 - Compute μ_s , compute $C_s(x)$, and store in a sorted array all numbers $m - \mu_s$ for exponents $m > \mu_s$ where $c_m \neq 0$.
- (2) Query step:
 - (a) For the SUBMASS QUERY PROBLEM: Search for each query mass M_i for $1 \leq i \leq k$, and return **yes** if found, **no** otherwise.

² Incidentally, our two different uses of "[]" are both standard, for generating functions and logical expressions, respectively. Since there is no danger of confusion, we have chosen to use both rather than introducing new ones.

(b) For the NUMBER OF SUBMASSES PROBLEM: Return size of array.

Correctness of ALGORITHM 1 follows immediately from the previous lemmas.

Theorem 2.3 ALGORITHM 1 *solves the SUBMASS QUERY PROBLEM in time $\mathcal{O}(\mu_s \log \mu_s + k \log n)$ and the NUMBER OF SUBMASSES PROBLEM in time $\mathcal{O}(\mu_s \log \mu_s)$.*

Proof: The polynomial $C_s(x)$ can be computed efficiently using Fast Fourier Transform (FFT)[16], which runs in time $\mathcal{O}(\mu_s \log \mu_s)$, since $C_s(x)$ has degree $2\mu_s$. Hence, the preprocessing step takes time $\mathcal{O}(\mu_s \log \mu_s)$. The query time for the SUBMASS QUERY PROBLEM is $\mathcal{O}(k \log \sigma(s)) = \mathcal{O}(k \log n)$. \square

Instead of using a sorted array, we can store the submasses in an array of size μ_s (which can be hashed to $\mathcal{O}(\sigma(s))$ size) and allow for direct access in constant time, thus reducing the query time to $\mathcal{O}(k)$.

As mentioned in the Introduction, we can employ methods from [2] for sparse polynomials and reduce $\deg C_s$ to $\mathcal{O}(\sigma(s))$, the number of non-zero coefficients. However, for the rest of this paper, we will refer to the running time as proportional to $\mu_s \log \mu_s$.

As an aside, note that $\mu_s \leq \mu_{\max} n$, where recall that $\mu_{\max} = \max \mu(\Sigma)$. If the maximal mass can be viewed as a constant, this yields runtime $\mathcal{O}(n \log n)$ for the preprocessing step. It may not always be realistic to assume that μ_{\max} is constant, because in order to enforce that all masses be positive integers, a scaling of the masses may be necessary, which can blow them up significantly.³ However, even in this case, the algorithm outperforms BINSEARCH for the SUBMASS QUERY PROBLEM as long as $\mu_{\max} = o(\frac{n}{\log n})$.

Along the same lines, for the NUMBER OF SUBMASSES PROBLEM, our algorithm allows computation of $\sigma(s)$ in $\mathcal{O}(\mu_s \log \mu_s) = \mathcal{O}(n \mu_{\max} \log(n \mu_{\max}))$ time. The naïve solution of generating all submasses requires $\Theta(n^2 \log n)$ time and $\Theta(\sigma(s))$ space (with sorting), or $\Theta(n^2)$ time and $\Theta(\mu_s)$ space (with an array of size μ_s). Our algorithm thus outperforms this naïve approach as long as $\mu_{\max} = o(\frac{n}{\log n})$.

³ This can be the case, e.g., for protein strings, where the amino acid masses are known up to a precision of more than 10^{-5} .

3 A Las Vegas Algorithm for Finding Witnesses

We now describe how to efficiently find a witness for each submass of the string s . Our high level idea is the following: We first note that given a mass M , if we know the ending position j of a witness of M , then, using the prefix masses p_1, \dots, p_n , we can easily find the beginning position of this witness. To do so, we simply do a binary search amongst the prefix masses p_1, \dots, p_{j-1} for mass $p_j - M$. Below, we will define two suitable polynomials of degree at most μ_s such that the coefficient of $x^{M+\mu_s}$ in their product equals the sum of the ending positions of substrings that have mass M .

Now, if we knew that there was a unique witness of mass M , then the coefficient would equal the ending position of this witness. However, this need not always be the case. In particular, if there are many witnesses with mass M , then we would need to check all partitions of the coefficient of $x^{M+\mu_s}$, which is computationally far too costly. To get around this problem, we look for the witnesses of M in the string s , where we do not consider all pairs of positions but instead random subsets of these.

By using the definition of $Q(x)$ from (2), set

$$R_s(x) := \sum_{i=1}^n i \cdot x^{p_i} \quad \text{and} \quad (4)$$

$$F_s(x) := R_s(x) \cdot Q_s(x) = \sum_{m=0}^{2\mu_s} f_m x^m. \quad (5)$$

In the following lemma, we use the definition of c_m from (3).

Lemma 3.1 *Let $m > \mu_s$. If $c_m = 1$, then f_m equals the ending position of the (sole) witness of $m - \mu_s$.*

Proof: By definition,

$$f_m = \sum_{(i,j) \text{ witness of } m} j$$

for any $m > \mu_s$. If $c_m = 1$, then, by Lemma 2.1, $m - \mu_s$ has exactly one witness (i_0, j_0) . Thus, $f_m = j_0$. \square

Example 2 *We continue with Example 1 on string $s = baac$ and masses 2, 3, 5 for characters a, b, c . We get $R_s(x) = x^3 + 2x^5 + 3x^7 + 4x^{12}$ and $F_s(x) = x^8 + 3x^{10} + 6x^{12} + 5x^{14} + x^{15} + 3x^{16} + 6x^{17} + 7x^{19} + 4x^{21} + 4x^{24}$. By checking the coefficients of $C_s(x)$, we see that among the exponents $m > \mu_s$, c_{15}, c_{16}, c_{21} , and c_{24} equal 1. Thus, with $F_s(x)$, we now know that the only witnesses of the submasses 3, 4, 9, and 12 end at positions 1, 3, 4, and 4, respectively.*

3.1 The Algorithm

We now present a Las Vegas algorithm for the SUBMASS WITNESS PROBLEM. In the algorithm, we first use polynomial $C_s(x)$ to generate a data structure containing all submasses of s . We then run a procedure which uses random subsets to try and find witnesses for each of these submasses. It outputs a set of pairs (m, j_m) , where m is a submass of s , and j_m is the ending position of one witness of m . Then, for each query mass which is in this set, we find the beginning position of the witness in time $\mathcal{O}(\log n)$ with binary search within the prefix masses, as described above. For any remaining query masses which are submasses of s , we simply run LINSEARCH to find a witness.

ALGORITHM 2

- (1) Compute $C_s(x)$ from Equation (3), and store all submasses of s .
- (2) Procedure TRY-FOR-WITNESS
 - (i) For a from 1 to $2 \log_2 n$, do:
 - (ii) Let $b = 2^{-a/2}$. Repeat $24 \ln n$ times:
 - (iii)
 - Generate a random subset I_1 of $\{1, 2, \dots, n\}$, and a random subset I_2 of $\{0, 1, 2, \dots, n-1\}$, where each element is chosen independently with probability b .
 - Compute $P_{I_1}(x) = \sum_{i \in I_1} x^{p_i}$, $Q_{I_2}(x) = \sum_{i \in I_2} x^{\mu_s - p_i}$ and $R_{I_1}(x) = \sum_{i \in I_1} i \cdot x^{p_i}$.
 - Compute $C_{I_1, I_2}(x) = P_{I_1}(x) \cdot Q_{I_2}(x)$ and $F_{I_1, I_2}(x) = R_{I_1}(x) \cdot Q_{I_2}(x)$.
 - Let $c_m = [x^m]C_{I_1, I_2}(x)$ and $f_m = [x^m]F_{I_1, I_2}(x)$.
 - For $m > \mu_s$, if $c_m = 1$ and if m has not yet been successful, then store the pair $(m - \mu_s, f_m)$. Mark m as successful.
- (3) Check which of the query masses is a submass of s by looking them up in the data structure generated in Step 1. Exclude all queries that are not submasses of s .
- (4) For all submasses amongst the queries M_ℓ , $1 \leq \ell \leq k$, which are marked as successful (i.e., an ending position was found by procedure TRY-FOR-WITNESS), find the beginning position with binary search amongst the prefix masses.
- (5) If there is a submass M_ℓ for which no witness was found, find one using LINSEARCH.

3.2 Analysis

We first give an upper bound on the failure probability of procedure TRY-FOR-WITNESS for a particular query mass M .

Lemma 3.2 *For a query mass M with $\kappa(M) = \kappa$, and $a = \lceil \log_2 \kappa \rceil$, consider the Step 2.iii of ALGORITHM 2. The probability that the coefficient $c_{M+\mu_s}$ of $C_{I_1, I_2}(x)$ for value a (as defined above) is not 1 is at most $\frac{7}{8}$.*

Proof: Let the witnesses of M be $\{(b_1, e_1), \dots, (b_\kappa, e_\kappa)\}$. Clearly $0 \leq \kappa \leq n$. We first analyze the probability of the event that for this particular choice of a , the coefficient of x^{μ_s+M} in $C(x)$ is exactly 1. This is the case if and only if $|\{i : b_i \in I_1 \text{ and } e_i \in I_2, 1 \leq i \leq \kappa\}| = 1$. Now, for $1 \leq i \leq \kappa$, let E_i denote the event $E_i = \{b_i \in I_1\} \cap \{e_i \in I_2\}$. Since for any $i \neq j$, we have $b_i \neq b_j$ and $e_i \neq e_j$, it follows that E_i and E_j are independent events. Thus, the probability that exactly one of the E_i 's holds is

$$\kappa 2^{-a} \cdot (1 - 2^{-a})^{\kappa-1} > \kappa 2^{-a} \cdot (1 - 2^{-a})^{2^a} \geq \frac{1}{2} \cdot \frac{1}{4} = \frac{1}{8}.$$

The last inequality follows because $(1 - \epsilon)^{1/\epsilon} \geq \frac{1}{4}$ for any $\epsilon \leq \frac{1}{2}$. \square

Lemma 3.3 *Procedure TRY-FOR-WITNESS does not find a witness for a given submass M with probability at most $1/n^3$. Moreover, the probability that the procedure fails for some submass is at most $1/n$.*

Proof: By Lemma 3.2 we know that for any fixed submass M , and for the particular choice of a , the probability that the random choice of I_1 and I_2 produces a unique witness for M is at least $1/8$. Since Step 2.iii is repeated $24 \ln n$ times, and all trials are independent of each other, the probability that there is no unique witness for any run is at most $(7/8)^{24 \ln n} \leq e^{-3 \log n} = \frac{1}{n^3}$. This follows since $(1 - \epsilon)^{1/\epsilon} \leq e^{-1}$ for any $0 < \epsilon < 1$. Since there are at most $O(n^2)$ different submasses in a string of length n , using the union bound, the algorithm generates a witness for each distinct submass with probability at least $1 - n^2 \cdot \frac{1}{n^3} = 1 - 1/n$. \square

Theorem 3.4 *ALGORITHM 2 solves the SUBMASS WITNESS PROBLEM in expected time $\mathcal{O}(\mu_s \log^3 \mu_s + k \log n)$.*

Proof: Denote the number of distinct submasses amongst the query masses by k' , thus, $k' \leq k$. By Lemma 3.3, the probability that procedure TRY-FOR-WITNESS finds a witness for each of the $k' = O(n^2)$ submasses is at least $1 - 1/n$. In this case, the running time is the time for running the procedure, plus the time for finding witness beginning positions. On the other hand, the

probability that the procedure fails to find a witness is at most $1/n$. In this case, we run `LINSEARCH` for the missing query masses, each in time $\mathcal{O}(n)$, thus at most in overall time $\mathcal{O}(k'n)$. Plugging it all together we get:

$$\begin{aligned}
& \mathcal{O}(\underbrace{\mu_s \log \mu_s}_{\text{Step 1.}} + \underbrace{2 \log n}_{\text{Step 2.i}} \cdot \underbrace{24 \ln n}_{\text{Step 2.ii}} \cdot \underbrace{\mu_s \log \mu_s}_{\text{Steps 2.iii}}) \\
+ & \underbrace{\mathcal{O}(k \log n)}_{\text{Step 3.}} + (1 - \frac{1}{n})\mathcal{O}(k' \log n) + \frac{1}{n}\mathcal{O}(k'n) \\
= & \mathcal{O}(\mu_s \log^3 \mu_s + k \cdot \log n).
\end{aligned}$$

□

4 A Deterministic Algorithm for Finding All Witnesses

Recall that, given the string s of length n and k query masses M_1, \dots, M_k , we are able to solve the `SUBMASS ALL WITNESSES PROBLEM` in $\Theta(k \cdot n)$ time and $\mathcal{O}(1)$ space with `LINSEARCH`, or in $\Theta(n^2 \log n + k \log n)$ time and $\Theta(n^2)$ space with `BINSEARCH`. Thus, the two naïve algorithms yield a runtime of $\Theta(\min(kn, (n^2 + k) \log n))$.

Our goal here is to give an algorithm which outperforms the bound above, provided certain conditions hold. Clearly, in general it is impossible to beat the bound $\min(kn, n^2)$ because that might be the size of the output, K , the total number of witnesses to be returned. Our goal will be to produce something good if $K \ll kn$.

First, consider two strings s and t and their concatenation $s \cdot t$. We are interested in submasses of $s \cdot t$ with a witness which spans or touches the border between s and t . More precisely, we refer to a witness (i, j) as a *border-spanning* witness if and only if $i \leq |s| \leq j$. We can encode such witnesses again in a polynomial, using the definition of $P(x)$ from (1). The idea is that the mass of a border-spanning witness can be written as the sum of a prefix mass of s^r , the reverse string of s , and a prefix mass of t . Note that here, we also allow 0 as a submass.

Lemma 4.1 *For two strings s, t , and the polynomial*

$$D_{s,t}(x) := (x^0 + P_{s^r}(x)) \cdot (x^0 + P_t(x)) = \sum_{m=0}^{\mu(s)+\mu(t)} d_m x^m, \quad (6)$$

the coefficient d_m equals the number of border-spanning witnesses of m in $s \cdot t$.

Proof: Straightforward. □

Example 3 Let $s = ba, t = ac$, and the masses as before. We get $D_{s,t}(x) = x^0 + 2x^2 + x^4 + x^5 + 2x^7 + x^9 + x^{12}$. We compare these to the terms of $\frac{1}{x^{12}}C_{baac}(x)$ with positive exponent in Example 1, since these yield all non-zero submasses of $baac$: $2x^2 + x^3 + x^4 + 2x^5 + 2x^7 + x^9 + x^{12}$. We see that the (sole) witness of 3 and one of the witnesses of 5 are not border-spanning witnesses.

4.1 The Algorithm

The algorithm combines the polynomial method with LINSEARCH in the following way: We divide the string s into g substrings of approximately equal length. We then use polynomials to identify, for each query mass M and each witness (b, e) of M , which substrings the beginning and end index lie in. Then we use LINSEARCH on these substrings to actually find the witnesses. The crucial observation is given in Lemma 4.2. We now describe the details.

We divide the string s into g substrings of approximately equal length: $s = t_1 \cdot t_2 \cdots t_g$ (where we will choose g below), and denote by $M_{i,j} = \sum_{m=i+1}^{j-1} \mu(t_m)$. In particular, if $j \leq i + 1$, then $M_{i,j} = 0$.

In order to have a good choice for g , we need to know the total size of the output, $K = \sum_{\ell=1}^k \kappa(M_\ell)$. This we can obtain by computing $C_s(x)$ and then adding up the coefficients $c_{M_\ell + \mu_s}$ for $1 \leq \ell \leq k$. We now set $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$. Observe that if $Kn \leq \mu_s \log \mu_s$, then $g = 1$, in which case we are better off running LINSEARCH. So let $Kn > \mu_s \log \mu_s$.

In Step 2.(b) of the following algorithm, we modify LINSEARCH to only return border-spanning submasses. This can be easily done by setting the second pointer at the start of the algorithm to the last position of the first string, and by breaking when the first pointer moves past the first position of the second string.

ALGORITHM 3

- (1) Preprocessing step:
 - (a) Compute μ_s and $C_s(x)$ as defined in (3), and compute $K = \sum_{\ell=1}^k c_{M_\ell + \mu_s}$. Set $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$.
 - (b) For each $1 \leq i \leq g$, compute $C_{t_i}(x)$.
 - (c) For each $1 \leq i < j \leq g$, compute $D_{t_i, t_j}(x)$ as defined in (6).
- (2) Query step:

- (a) Compute a witness-position-list for each query M_ℓ by iterating through all terms of the C_{t_i} 's and all terms of the D_{t_i, t_j} 's. The witness-position-list of M_ℓ contains exactly those i such that M_ℓ is a submass of t_i , and those pairs (i, j) , such that $M_\ell - M_{i,j}$ is a border-spanning submass of $t_i \cdot t_j$.
- (b) For each $1 \leq \ell \leq k$,
 - (i) If M_ℓ 's witness-position-list is empty, then return no.
 - (ii) For each i in M_ℓ 's witness-position-list, run LINSEARCH on t_i for M_ℓ and return all witnesses.
 - (iii) For each pair (i, j) in M_ℓ 's witness-position-list, run LINSEARCH on $t_i \cdot t_j$ for submass $M_\ell - M_{i,j}$ and return all border-spanning witnesses.

4.2 Analysis

The following lemma shows the correctness of ALGORITHM 3.

Lemma 4.2 For $1 \leq M \leq \mu_s$,

$$\kappa(M) = \sum_{i=1}^g [x^{M+\mu(t_i)}]C_{t_i} + \sum_{1 \leq i < j \leq g} [x^{M-M_{i,j}}]D_{t_i, t_j}(x).$$

Proof: First, observe that for any witness (b, e) of M , there is exactly one pair (i, j) such that b lies in string t_i and e in t_j . If $i = j$, then M is a submass of t_i and by Lemma 2.1 contributes exactly 1 to the coefficient $[x^{M+\mu(t_i)}]C_{t_i}(x)$. Otherwise, $i < j$, and $M - M_{i,j}$ is a submass of the concatenated string $t_i \cdot t_j$ with the witness (b', e') , where (b', e') is shifted appropriately (i.e., $b' = b - \sum_{i' < i} |t_{i'}|$ and $e' = e - \sum_{i' < j} |t_{i'}|$). Moreover, (b', e') is a border-spanning submass of $t_i \cdot t_j$. Thus, by Lemma 4.1, (b', e') contributes exactly 1 to $[x^{M-M_{i,j}}]D_{t_i, t_j}(x)$. \square

For the runtime analysis of ALGORITHM 3, we first show that the preprocessing step of ALGORITHM 3 has runtime $\mathcal{O}(g\mu_s \log \mu_s)$. To see this, observe that the time for computing the polynomials with FFT is

$$\begin{aligned} \text{for } C_s(x): & \quad \mathcal{O}(\mu_s \log \mu_s), \\ \text{for the } C_{t_i}(x)\text{'s:} & \quad \mathcal{O}\left(\sum_{i=1}^g \mu(t_i) \log(\mu(t_i))\right), \\ \text{for the } D_{i,j}(x)\text{'s:} & \quad \mathcal{O}\left(\sum_{1 \leq i < j \leq g} (\mu(t_i) + \mu(t_j)) \log(\mu(t_i) + \mu(t_j))\right). \end{aligned}$$

Together, the terms above yield

$$\begin{aligned} & \mathcal{O}\left(\underbrace{\mu_s \log \mu_s + \sum_{i=1}^g \mu(t_i) \log(\mu(t_i))}_{\leq \mu_s \log \mu_s} + \underbrace{\sum_{1 \leq i < j \leq g} (\mu(t_i) + \mu(t_j)) \log(\mu(t_i) + \mu(t_j))}_{\leq g \mu_s \log \mu_s}\right) \\ &= \mathcal{O}(g \mu_s \log \mu_s). \end{aligned}$$

For the upper bound on the third term, note that $\sum_{1 \leq i < j \leq g} (\mu(t_i) + \mu(t_j)) = \sum_{i=1}^g (g-1) \cdot \mu(t_i) = (g-1) \mu_s$.

Now for the query time of ALGORITHM 3: First, in Step 2a, we compute for each query M_ℓ the witness-position-list that contains all i s.t. $[x^{M_\ell + \mu(t_i)}]C_{t_i}(x) \neq 0$, and all (i, j) s.t. $[x^{M_\ell - M_{i,j}}]D_{t_i, t_j}(x) \neq 0$. These lists can be computed by iterating first through all non-zero coefficients c_m of each C_{t_i} , $1 \leq i \leq g$, and checking whether $m + \mu(t_i)$ is among the query masses. Recall that there are $\mathcal{O}(\mu_s \log \mu_s)$ many of these coefficients. Next, we iterate through all non-zero coefficients d_m of each D_{t_i, t_j} , $1 \leq i < j \leq g$, and check whether $m + M_{i,j}$ is among the query masses. Again, there are $\mathcal{O}(g \mu_s \log \mu_s)$ many coefficients to check. Together, we get a runtime of $\mathcal{O}(g \mu_s \log \mu_s)$ if we have constant access to the query masses, or $\mathcal{O}(g \mu_s \log \mu_s \cdot \log k)$ if they are stored in a binary array.

Now, in step 2b, for each query mass M_ℓ , we run LINSEARCH for each entry in the witness-position-list, thus at most $\kappa(M_\ell)$ many times. The LINSEARCH step for one entry takes at most $2n/g$ time. Thus, we get query time $\mathcal{O}(g \mu_s \log \mu_s + K \frac{n}{g})$. With $g = \lceil (\frac{Kn}{\mu_s \log \mu_s})^{\frac{1}{2}} \rceil$, the total runtime becomes $\mathcal{O}((Kn \mu_s \log \mu_s)^{\frac{1}{2}})$, and we have thus proved the following theorem:

Theorem 4.3 ALGORITHM 3 solves the SUBMASS ALL WITNESSES PROBLEM in time $\mathcal{O}((Kn \mu_s \log \mu_s)^{\frac{1}{2}})$, where K is the total number of witnesses, i.e., the output size.

To better understand this result, let $\bar{\kappa}$ denote the average size of the output, i.e., $\bar{\kappa} = K/k$. Then the runtime is $(k \bar{\kappa} n \mu_s \log \mu_s)^{1/2}$. Recall that the running time of the combination of the naïve algorithms for the submass all witnesses problem is $\mathcal{O}(\min(kn, n^2 \log n))$. Thus, our algorithm beats the running time of the naïve algorithms above if $\bar{\kappa} \mu_s \log \mu_s = o(kn)$ and $(\bar{\kappa} k \mu_s \log \mu_s) = o(n^3 \log^2 n)$.

4.3 A Variation for One Witness per Query

ALGORITHM 3 can be straightforwardly adapted to only produce one witness per query mass, i.e., to solve the SUBMASS WITNESS PROBLEM. Then

its runtime becomes $\mathcal{O}((kn\mu_s \log \mu_s)^{\frac{1}{2}})$, i.e., somewhere between $\mathcal{O}(\mu_s \log \mu_s)$ and $\mathcal{O}(kn)$ (since kn needs to be the larger factor if we want to employ the algorithm). If, say, $k = \mathcal{O}(\mu_s)$, then we end up with a runtime of $\mathcal{O}(\mu_s \sqrt{n})$. Comparing this to the $\mathcal{O}(\mu_s \text{polylog } \mu_s)$ runtime of ALGORITHM 2 leaves us with an extra \sqrt{n} factor which we pay for the deterministic version.

5 Conclusion

In this paper we gave algorithms for several variants of finding substrings with given masses in a given weighted string (the Submass Finding Problem). Our algorithms are most interesting when the masses of the individual characters are small compared to the length of the string, or more generally, when the number of different possible submasses is small compared to n^2 .

Most of our algorithms have running time complexity dependent (up to polylog factors) on the number of different submasses in the given weighted string. While this may not be the best possible running time, it seems that improving this significantly will be hard. For example, consider the problem of finding the number of different submasses $\sigma(s)$. Our algorithm for this problem has runtime $\mathcal{O}(\sigma(s) \log \sigma(s))$. On the other hand, the easier problem of *deciding* whether $\sigma(s)$ is exactly equal to $n(n+1)/2$ is already at least as hard as the 4-Sum problem. To see this, let p_i denote the i 'th prefix mass as before (i.e., the mass of the first i characters of the string s); then $\sigma(s) < n(n+1)/2$ if and only if there are distinct integers i, j, k, l such that $p_i - p_j = p_k - p_l$, which is exactly the 4-Sum problem. The 4-Sum problem is conjectured to have a runtime complexity of $\Omega(n^2)$ [17,18] and is one of the major problems in computational geometry. So, it is unlikely that even the number of different submasses can be determined in time $o(n^2)$ in the general case.

Acknowledgments A preliminary version of this work is due to appear in the Proceedings of the Fifteenth Annual Combinatorial Pattern Matching Symposium (CPM 2004).

References

- [1] H. Wilf, *generatingfunctionology*, Academic Press, 1990.
- [2] R. Cole, R. Hariharan, Verifying candidate matches in sparse and wildcard matching, in: Proc. of the 34th Symposium on the Theory of Computing (STOC), 2002, pp. 592–601.

- [3] A. G. Murzin, S. E. Brenner, T. J. P. Hubbard, C. Chothia, SCOP: a structural classification of proteins database for the investigation of sequences and structures, *Journal of Molecular Biology* 247 (1995) 536–540.
- [4] A. G. Murzin, L. L. Conte, A. Andreeva, D. Howorth, B. G. Ailey, S. E. Brenner, T. J. P. Hubbard, C. Chothia, Structural classification of proteins, <http://scop.mrc-lmb.cam.ac.uk/scop/> (2004).
- [5] M. Cieliebak, T. Erlebach, Z. Lipták, J. Stoye, E. Welzl, Algorithmic complexity of protein identification: Combinatorics of weighted strings, *Discrete Applied Mathematics (DAM)* 137 (1) (2004) 27–46.
- [6] N. Edwards, R. Lippert, Generating peptide candidates from amino-acid sequence databases for protein identification via mass spectrometry, in: *Proc. of the 2nd International Workshop on Algorithms in Bioinformatics (WABI), 2002*, pp. 68–81.
- [7] J. Eng, A. McCormack, J. R. Yates III, An approach to correlate tandem mass spectral data of peptides with amino acid sequences in a protein database, *Journal of the American Society for Mass Spectrometry (JASMS)*. 5 (1994) 976–989.
- [8] B. Lu, T. Chen, A suffix tree approach to the interpretation of tandem mass spectra: Applications to peptides of non-specific digestion and post-translational modifications, *Bioinformatics, Supplement 2 (ECCB) (2003)* ii113–ii121.
- [9] G. Benson, Composition alignment, in: *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI), 2003*, pp. 447–461.
- [10] S. Böcker, Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt, in: *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI), 2003*, pp. 476–497.
- [11] S. Böcker, SNP and mutation discovery using base-specific cleavage and MALDI-TOF mass spectrometry, *Bioinformatics, Supplement 1 (ISMB) (2003)* i44–i53.
- [12] A. Salomaa, Counting (scattered) subwords, *Bulletin of the European Association for Theoretical Computer Science (EATCS)* 81 (2003) 165–179.
- [13] R. Eres, G. M. Landau, L. Parida, A combinatorial approach to automatic discovery of cluster-patterns, in: *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI), 2003*, pp. 139–150.
- [14] A. Amir, A. Apostolico, G. Landau, G. Satta, Efficient text fingerprinting via Parikh mapping, *Journal of Discrete Algorithms* 1 (5-6) (2003) 409–421.
- [15] G. Didier, Common intervals of two sequences, in: *Proc. of the 3rd International Workshop on Algorithms in Bioinformatics (WABI), 2003*, pp. 17–24.
- [16] J. W. Cooley, J. W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Mathematics of Computation* 19 (90) (1965) 297–301.

- [17] E. D. Demaine, J. S. B. Mitchell, J. O'Rourke, The open problems project, <http://cs.smith.edu/~orourke/TOPP/> (2004).
- [18] J. Erickson, Lower bounds for linear satisfiability problems, in: Proc. of 6th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1995, pp. 388-395.